Acta Cybernetica **18** (2008) 651–668.

Factored Temporal Difference Learning in the New Ties Environment*

Viktor Gyenes†, Ákos Bontovics†, and András Lőrincz†‡

Abstract

Although reinforcement learning is a popular method for training an agent for decision making based on rewards, well studied tabular methods are not applicable for large, realistic problems. In this paper, we experiment with a factored version of temporal difference learning, which boils down to a linear function approximation scheme utilising natural features coming from the structure of the task. We conducted experiments in the New Ties environment, which is a novel platform for multi-agent simulations. We show that learning utilising a factored representation is effective even in large state spaces, furthermore it outperforms tabular methods even in smaller problems both in learning speed and stability, because of its generalisation capabilities.

Keywords: reinforcement learning, temporal difference, factored MDP

1 Introduction

Reinforcement learning (RL) [16] is a framework for training an agent for a given task based on positive or negative feedback called *immediate rewards* that the agent receives in response to its actions. Mathematically, the behaviour of the agent is characterised by a *Markov decision process* (MDP), which involves the *states* the agent can be in, *actions* the agent can execute depending on the state, a *state transition model*, and the *rewards* the agent receives.

For small, discrete state spaces well-studied tabular methods exist for solving the learning task. However, real world tasks include many variables, often continuous ones, for which the state space is very large, or even infinite, making these

*This material is based upon work supported partially by the European Office of Aerospace Research and Development, Air Force Office of Scientific Research, Air Force Research Laboratory, under Contract No. FA-073029. This research has also been supported by an EC FET grant, the 'New Ties project' under contract 003752. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the European Office of Aerospace Research and Development, Air Force Office of Scientific Research, Air Force Research Laboratory, the EC, or other members of the EC New Ties project.

†Eötvös Loránd University, Department of Information Systems, E-mail: gyenesvi@inf.elte.hu, bontovic@elte.hu, andras.lorincz@elte.hu

‡Corresponding author

approaches inapplicable. Consider a sequential decision problem with m variables. In general, we need an exponentially large state space to model it as an MDP. Thus, no matter if the solution time is polynomial in the number of states, scaling is *exponential* in the size of the task description (m). Fortunately, by exploiting state space structure, it is possible to drastically reduce the time needed for solving large MDPs. It is often the case that the transition model for a given variable depends only on a few other variables. Also, rewards may be associated with only a small number of variables. In such situations, *factored* MDPs (fMDPs) offer a more compact representation by taking the state space as the Cartesian product of m variables: $X = X_1 \times X_2 \times \ldots \times X_m$. This gives rise to learning in a factored manner, by separating variables as much as possible, taking into account (a priori known) dependencies among them and circumventing combinatorial explosion.

One popular method for solving MDPs is based on *value functions* that represent long term utilities that can be collected starting from a given state. The agent's behaviour, also called *policy*, is then defined by acting greedily according to this value function, i.e. selecting actions that result in next states with highest possible value. For factored problems one may be able to define the value function as the sum of *local scope functions*, i.e. functions that depend only on a small number of variables. This way, both the memory capacity needed to store the value function, and the learning time might be reduced by an exponential factor.

Many algorithms exist for solving factored MDPs. For the *value iteration* method convergence proof for factored MDPs has been provided recently [17]. In this paper, we experiment with the more flexible *temporal difference learning* (TD) method, in which the agent updates state values it actually observes during interacting with the world, as opposed to value iteration, which iteratively updates all state values synchronously. The advantage of our method is known; value iteration requires a model, whereas no model is required for TD learning.

The rest of the paper is structured as follows: in Section 2 we review the basics of reinforcement learning, especially value iteration and TD learning, motivate factored MDPs and derive our factored TD learning algorithm. Section 3 provides an overview of other approaches to learning in factored MDPs. In Section 4 we show simulation results utilising the newly introduced factored TD learning algorithm. Section 5 discusses some decisions in the choice of our agent architecture and experimental setup, and finally Section 6 concludes the paper.

2 Factored Reinforcement Learning

2.1 Value-function based Reinforcement Learning

Consider an MDP characterised by the tuple (X, A, P, R, γ), where X is the (finite) set of states the agent can be in, A is the (finite) set of actions the agent can execute, $P : X \times A \times X \to [0, 1]$ is the transition probability model, i.e. $P(x, a, x')$ is the probability that the agent arrives at state x' when executing action a in state x, $R : X \times A \to \mathbb{R}$ is the reward function, and γ is the discount rate on future

rewards.

A (Markov) policy of the agent is a mapping $\pi : X \times A \to [0, 1]$ so that $\pi(x, a)$ tells the probability that the agent chooses action a in state x. For any $x_0 \in X$, the policy of the agent determines a stochastic process experienced through the instantiation

$$x_0, a_0, r_0, x_1, a_1, r_1, \ldots, x_t, a_t, r_t, \ldots$$

where r_i is the reward received after executing action a in state x_i. In value-function based reinforcement learning the agent maintains a value function $V : X \to \mathbb{R}$, which reflects the expected value of the discounted total rewards collected by starting from state x and following policy π:

$$V^\pi(x) := E\Big(\sum_{t=0}^{\infty} \gamma^t r_t \;\Big|\; x = x_0\Big) \; .$$

Let the optimal value function be

$$V^*(x) := \max_\pi V^\pi(x)$$

for each $x \in X$. If V^* is known, it is easy to find an optimal policy π^*, for which $V^{\pi^*} = V^*$. Value functions satisfy the famous Bellman equations:

$$V^\pi(x) = \sum_a \sum_{x'} \pi(x, a) P(x, a, x') \Big(R(x, a) + \gamma V^\pi(x')\Big) \tag{2.1}$$

and

$$V^*(x) = \max_a \sum_{x'} P(x, a, x') \Big(R(x, a) + \gamma V^*(x')\Big) \; . \tag{2.2}$$

An optimal policy can be defined by acting greedily according to V^*, that is, by selecting actions that maximise $V^*(x')$, the value of the next state:

$$a^*(x) \in \arg\max_a \sum_{x'} P(x, a, x') \Big(R(x, a) + \gamma V^*(x')\Big) \; .$$

One may also define a function of state-action values, or *Q-values*, expressing the expected value of the discounted total rewards collected by starting from state x and executing action a and following policy π onwards:

$$Q^\pi(x, a) = \sum_{x'} P(x, a, x') \Big(R(x, a) + \gamma V^\pi(x')\Big) \; .$$

Action selection then becomes $a^*(x) = \arg\max_a Q(s, a)$. It is also true that the optimal policy satisfies the following equation:

$$Q^*(x, a) = \sum_{x'} P(x, a, x') \Big(R(x, a) + \gamma \arg\max_{a'} Q^*(x', a')\Big) \; . \tag{2.3}$$

There are many alternatives to this setting. For example, one may use a reward function $R(x, x')$ depending on the current state x and the next state x', but not on the action executed. Also, one may replace action a by desired next state x_d, provided that there is an underlying sub-policy (not necessarily optimised) for reaching state x' from state x in one step [18].

Most algorithms that solve MDPs build on the Bellman equations [1]. In the following, we shall concentrate on the value iteration and the temporal difference learning algorithms. In tasks when the transition model P and the reward function R are not known a priori, the agent also needs to learn them on its own.

2.2 Value Iteration and Temporal Difference Learning

Value iteration uses the Bellman equations (2.2) as an iterative assignment. Starting from an arbitrary value function V_0 (represented as a table) it performs the update

$$V_{t+1}(x) := \max_a \sum_{x' \in X} P(x, a, x') \Big(R(x, a) + \gamma V_t(x') \Big)$$

for all $x \in X$. As it is well known (see e.g. [1]), the above update converges to a unique solution, and the solution satisfies the Bellman equations (2.2).

As this method updates *all* states at the same time, it is considered a *synchronous* algorithm. One *sampled, incremental* version of the algorithm can be obtained by updating only the values (by a small amount) of the states actually observed by the agent during its interaction with the environment, according to the following formula:

$$\begin{aligned} V_{t+1}(x) &:= (1 - \alpha) V_t(x) + \alpha \Big(R(x, a) + \gamma V_t(x') \Big) \\ &= V_t(x) + \alpha \Big(R(x, a) + \gamma V_t(x') - V_t(x) \Big) \\ &= V_t(x) + \alpha \delta_t \ , \end{aligned}$$

where α is an update rate and δ_t is the difference between the currently approximated value of the state x and its approximation based on the next state and the current reward, hence the name temporal difference. This is the so called temporal difference (TD) learning method.

The formula presented here is called the TD(0) method, as it only takes the immediate next state into account when updating the value of a state. It has been proven that this sampled version of the value function update is convergent. The method can be extended to longer time spans by means of the so called eligibility traces (TD(λ)) [16]. Note however, that TD learning is a policy evaluation method, thus it converges to the value function of a *fixed* policy, while value iteration converges the *optimal* value function. TD learning can be combined with optimistic policy iteration to guarantee convergence to the optimal value function (see [16]).

If the model parameters P and R are not known, they must also be learned from interaction with the environment; these functions are naturally sampled as

the agent interacts with the world. Estimation of the transition probabilities P can be performed by frequency counting, whereas the reward function R can be estimated by averaging if these quantities are represented as tables; i.e. separately for each state. Also, it is straightforward to rewrite the above update rules for Q values.

Unfortunately, despite their attractive convergence property, tabular methods are not applicable for real world tasks, where the state space is very large. The problem is that memory requirement grows exponentially with the number of the Cartesian variables and thus learning is slow. There is hope though: many of the states are similar from the point of view of the long term cumulated reward and one might try to generalise for similar states.

To this end, RL methods have been extended from tabular methods to function approximation to represent the value function (for an excellent introduction, see, e.g., [16]). For example, the value function can be expressed as linear combination of basis functions that characterise the states, or by more sophisticated techniques, including multi-layer neural networks. Unfortunately, convergence results are rare for function approximators: convergence can be proven for linear function approximators under certain conditions. Also, the quality of the function approximation heavily depends on the basis functions also called 'features'.

One thus prefers features that enable linear function approximation for the value functions *and* take advantage of the factorised nature of the state space. The value of a state variable in the successive time step is conditioned on the previous values of possibly a *few* other variables and the action taken. The reward function usually depends on a couple of variable combinations, i.e. the actual reward can be given by the actual value of a *few* variables. Knowledge about these dependencies enables us to construct relevant features for value function approximation, as it will be described in the next section.

2.3 Factored Markov Decision Processes

Consider a sequential decision process with m variables. In general, we need an exponentially large state space to model it as an MDP, thus the number of states is *exponential* in the size of the description of the task. Factored Markov decision processes offer a more compact task representation.

In the fMDP case one assumes that X is the Cartesian product of m smaller state spaces, corresponding to individual discrete variables having arbitrary (but finite) number of possible values:

$$X = X_1 \times X_2 \times \ldots \times X_m \ .$$

Continuous valued variables may also be fit into this frame by discretising them, and substituting with a discrete variable having as many possible values as the number of (disjoint) intervals used in the discretisation. The actual value of the new discrete variable become the index of the interval that the continuous value falls into.

Furthermore, let us call a function f a *local scope function* on X if f only depends on a (small) subset of the variables $\{X_i\}$. For the ease of notation, let us denote a set of variables by $X[I]$ and their corresponding instantiation by $x[I]$, where $I \subset \{1, \ldots, m\}$ is an index set.

Now, let us assume that we have an MDP on state space X. Then, the transition model of the factored MDP can be defined in a more compact manner than that of an MDP with states having no internal structure. The transition probability from one state to another can be obtained as the product of several simpler factors, by providing the transition probabilities for each variable X_i separately, depending on the previous values of itself and the other variables. In most cases, however, the next value of a variable does not depend on all of the variables; only on a few. Suppose that for each variable x_i there exist sets of indices Γ_i such that the value of x_i in the next time step depends only on the values of the variables $x[\Gamma_i]$ and the action a taken. Then we can write the transition probabilities in a factored form:

$$P(x, a, x') \equiv P(x' \mid x, a) = \prod_{i=1}^{m} P_i(x'_i \mid x[\Gamma_i], a)$$

for each $x, x' \in X$, $a \in A$, where each factor is a local-scope function

$$P_i : X[\Gamma_i] \times A \times X_i \to [0, 1] \qquad \text{for all } i \in \{1, \ldots, m\}. \tag{2.4}$$

By contrast, in the non-factored form, the probabilities of the components of x' can not be computed independently from subsets of all variables. Assuming that the number of variables of the local scope functions is small, then these functions can be stored in small tables. The size of these tables is a sharp (exponential) function of the number of variables in the Γ_i sets. These tables are essentially *conditional probability tables* of a dynamic Bayesian network (see e.g., [3]) of m variables.

The reward model of the factored MDP also assumes a more compact form provided that the reward function depends only on (the combination) of a few variables in the state space. Formally, the reward function is the sum of local-scope functions:

$$R(x, a) = \sum_{j=1}^{k} R_j(x[I_j], a),$$

with arbitrary (but preferably small) index sets I_j, and local-scope functions

$$R_j : X[I_j] \times A \to \mathbb{R} \qquad \text{for all } j \in \{1, \ldots, k\}.$$

The functions R_i might also be represented as small tables. If the maximum size of the appearing local scopes is bounded by some constant and independent of the number of variables m, then the description length of the factored MDP is polynomial disregard of the number of variables m.

To sum up, a factored Markov decision process is characterised by the parameters

$$\left(\{X_i\}_1^m \ , \ A, \ \{\Gamma_i : P_i\}_1^m \ , \ \{I_j : R_j\}_1^k \ , \ \gamma \right) .$$

The optimal value function can be represented by a table of size $\prod_{i=1}^{m} |X_i|$, one table entry for each state. To represent it more efficiently, we may rewrite it as the sum of local-scope functions with small domains. Unfortunately, in the general case, no exact factored form exists [8], however, we can still approximate the function by means of local scope functions:

$$\hat{V}(x) = \sum_{j=1}^{n} V_j(x[J_j]) \ . \tag{2.5}$$

The local scope functions V_j can be represented by tables of size $\prod_{i \in J_j} |X_i|$, which are small provided that the sets J_j are small, i.e., they involve only a few number of variables. The question is, how can we provide index sets J_j that are relevant for the approximation of the value function. If the local scopes Γ_i and I_j for the transition model and the reward model are known (which might be easy to define manually having sufficient knowledge about the task and the variables involved), we may use the following reasoning to deduce scopes for the value function: the value function is the long-term extended version of the reward function (whose index sets I_j are known). If we want to come up with an index set J_j of a local scope value function V_j which reflects long term values one step before reaching rewarding states, we need to examine which variables influence the variables in the set I_j. We can go on with this recursively to find ancestors of the variables in the set I_j, and iteratively determine the sets of variables that predict values on the long term. This process is called backprojection through the transition model [8].

We may rearrange the terms of the value function and redefine the linear approximation as follows. The table entries represent weights corresponding to *binary features*. Consider a local scope index set J_j, which means that the local value function V_j depends on the variables $\{X_i : i \in J_j\}$. Let $N_j = \prod_{i \in J_j} |X_i|$. We introduce binary features of the form $F_l(x) = \delta(\bigwedge_{i \in J_j} x_i = v_{l_i})$ for each possible value combination $\{(v_{l_1}, \ldots, v_{l_{|J_j|}}) \ , \ l = 1, \ldots, N_j\}$ of the variables $\{X_i : i \in J_j\}$. Function δ is the indicator function for condition c; $\delta(c) = 1$ if condition c is true, and 0 otherwise. That is, each table defines as many binary features as the size of the table. Then the value function approximation can be rewritten as:

$$\hat{V}(x) = \sum_{l=1}^{N} w_l F_l(x), \tag{2.6}$$

where $N = \sum_{j=1}^{n} N_j$, by reindexing the features F_l to run from 1 to N.

This form enables us to employ reinforcement learning techniques developed for linear function approximators. In this paper, we use (2.6) and apply temporal difference learning to factored Markov decision processes. The update of the parameters w is based on gradient descent utilising the temporal difference δ_t:

$$w_{t+1} = w_t + \alpha \delta_t \nabla_w V(x_t) = w_t + \alpha \delta_t F(x_t) \ , \tag{2.7}$$

where $F(x_t)$ is the vector of binary feature values for state x_t, α is the update rate. Eligibility traces and TD(λ) learning techniques are also applicable to linear function approximation (see [16] for an introduction).

It has been shown that the synchronous version of factored value iteration is convergent [17]. Moreover, when sampling is applied, the algorithm requires only a number of samples polynomial in the number of variables m (which can be much smaller than the number of states) to approximate the value function well with high probability. We expect – based on the close relationship between factored value iteration and factored temporal difference learning – that tabular temporal difference learning is also convergent, although we can not prove at the moment that convergence results for factored value iteration could be transferred to factored temporal difference learning.

When the factored model parameters, i.e. local scope functions P_i and R_i are unknown, they can be approximated from experience. The conditional probability tables corresponding to the local scope functions P_i can be updated separately by frequency counting for all variables and actions when observing state-to-state transitions. The factored reward function R can also be thought of as a linear function approximator, for example $R(x,a) = \sum_l u_l G_l(x,a)$ or $R(x,x') = \sum_l u_l G_l(x,x')$ based on some binary features G_l and parameters u_l (similarly to the value function), and can be updated using standard gradient descent techniques.

3 Related work

The exact solution of factored MDPs is infeasible. The idea of representing a large MDP using a factored model was first proposed by Koller and Parr [9]. More recently, the framework (and some of the algorithms) was extended to fMDPs with hybrid continuous-discrete variables [10] and factored partially observable MDPs [13]. Furthermore, the framework has also been applied to structured MDPs with alternative representations, e.g., relational MDPs [7] and first-order MDPs [14].

There are two major branches of algorithms for solving fMDPs: the first one approximates the value functions as decision trees, the other one makes use of linear programming.

Decision trees (or equivalently, decision lists) provide a way to represent the agent's policy compactly. Algorithms to evaluate and improve such policies, according to the policy iteration scheme have been worked out in the literature (see [9] and [2, 3]). Unfortunately, the size of the policies may grow exponentially even with a decision tree representation [3, 11].

The exact Bellman equations (2.2) can be transformed to an equivalent linear program with N variables $\{V(x) : x \in X\}$ and $N \cdot |A|$ constraints. In the approximate linear programming approach, we approximate the value function as a linear combination of basis functions (see, (2.5)), resulting in an approximate LP (ALP) with n variables $\{w_j : 1 \leq j \leq n\}$ and $N \cdot |A|$ constraints. Both the objective function and the constraints can be written in compact forms, exploiting the local-scope property of the appearing functions.

Markov decision processes were first formulated as LP tasks by Schweitzer [15]. The approximate LP form is a work of Farias [4]. Guestrin [8] shows that the maximum of local-scope functions can be computed by rephrasing the task as a non-

serial dynamic programming task and eliminating variables one by one. Therefore, the ALP can be transformed to an equivalent, more compact linear program. The gain may be exponential, but this is not necessary in all cases. Furthermore, the cost of the transformation may scale exponentially [5].

The approximate LP-based solution algorithm was worked out by Guestrin [8]. Primal-dual approximation technique to the linear program is applied by Dolgov [6], and improved results on several problems are reported.

The approximate policy iteration algorithm [9, 8] also uses an approximate LP reformulation, but it is based on the policy-evaluation Bellman equation (2.1). Policy-evaluation equations are, however, linear and do not contain the maximum operator, so there is no need for the second, costly transformation step. On the other hand, the algorithm needs an explicit decision tree representation of the policy. Liberatore [11] has shown that the size of the decision tree representation can grow exponentially.

4 Experiments

4.1 The scenario

The experiments reported in this paper were performed in a grid-world environment. This environment is part of an EC FET project, the 'New Ties project', which is a novel platform for multi-agent simulations. In the present simulations we experimented with single agents in order to evaluate our learning mechanisms, but the factored technique enables us to consider multi-agent scenarios in the future: agents can compute optimal behaviour by approximating other agents as additional factors.

The rectangular grid world contained two groups of food items at the far ends of the world. The task of the agent was to learn to consume food appropriately to survive: keep its energy level between two thresholds, that is, avoid being hungry, but also avoid being too much full; it received punishment for having the energy in the wrong ranges. Table 1 summarises the rewards of the agent depending on its energy level.

energy level	$\Delta_E < 0$	$\Delta_E = 0$	$\Delta_E > 0$
below lower threshold	-1	-1	Δ_E
in appropriate range	Δ_E	0	0
above upper threshold	$-\Delta_E$	-1	-1

Table 1: **Numerical values of the rewards.** Δ_E: change in the agent's energy. Additional component of the reward: cost for the agent's distance from home changed linearly in the range [0, 0.1].

The agent had a so called metabolism, so that it was better for the agent to consume both kind of food items, that is, if the agent consumed only one kind of

food, then its energy did not increase after a while. Also, we augmented the task with punishments for being far away from home, where home of the agent was its starting position in the grid world.

The agent was only able to observe the world partially, i.e. it had a cone of sight in front of it with a limited range. The agent moved on an 8-neighbourhood grid; it was able to turn left or right 45 degrees, and move forward. It had a cone of sight of 90 degrees in front of itself. It had a 'bag' of limited capacity, into which it was able to collect food items, and later consume the food from the bag. The primitive observations of the agent were food items in its cone of sight, its own level of energy and the number of food items in its bag of each type. The primitive actions were 'turning left/right', 'moving forward', 'picking up food to the bag', and 'eating food from the bag'.

4.2 Agent architecture

Since reinforcement learning in a heavily partially observable environment is very difficult in general and because the Markovian assumption on the state description is not met, we augmented the agent with high level variables and actions in order to transform the task and improve its Markovian properties. We note that there are formal approaches to tackle the problem of partial observability that aim to transform the series of observations automatically into a Markovian state description via belief states (see, e.g., [12] and the references therein), we did not choose to utilise the framework in the present study for the following reasons. First, we aimed to separate the factored MDP approach in a demanding scenario from the demands of partial observability. Second, the generation of high level features (state variables) from low level observations is a great challenge for artificial intelligence and is far from being solved in general and we wanted to gain insight into this problem through the scenarios. We do not list our negative experiences here, although they might be as important as the solution that we describe below.

The predefined high level variables were calculated from the history of observations and formed the variables of the state space of the factored MDP. The history of observations were stored using so called long term *memory maps*, for example one containing entries about where the agent has seen food items of a certain type in the past. Also, high level action macros were manually programmed as a series of primitive actions to facilitate navigation at a higher level of abstraction.

Figure 1 shows our agent architecture that makes use of high level variables and actions, and the factored architecture for value function approximation.

A sketch of the functioning of the agent architecture is shown in Algorithm 1. In the core of the algorithm is temporal difference learning with function approximation, for which memory maps serve as a preprocessing step to generate the current state of the agent from the history of observations. The agent also performs state transition and reward model learning.

Table 2 enumerates the high level variables and action macros that we used. In most cases the macros are related to variables; they can be chosen by the agent to alter the values of the variables, thus they are shown side by side in the table.

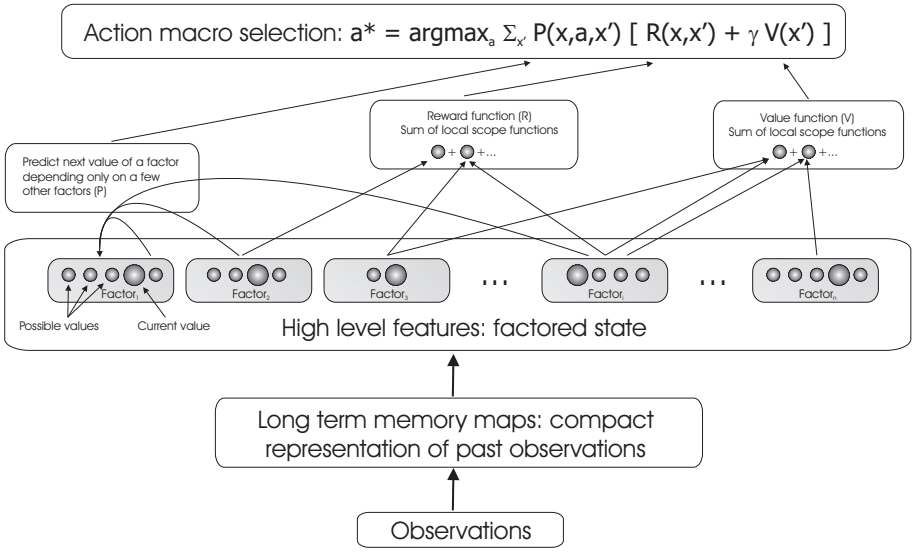


Figure 1: **Agent architecture.** The history of observations is summarised by long term memory maps. Based on these past and present observations, high level variables are generated, which form the variables of the factored MDP. The transition model (P) is learned as the product of local scope functions, and the reward (R) and value functions (V) are learned as the sum of local scope functions. Utilising these functions, action macro selection is accomplished in a greedy manner.

To make the description complete, we also need to provide the scopes of the local scope functions used. For the transition probabilities, this means providing the variables each state variable depends on, considering its next value when executing an action. For most variables, its next value depended only on its own previous value and the action taken, except for the energy level, which depended on itself, and the food history features as well. The reward function had factors depending on the energy level and the distance from home. The value function, which expresses long term cumulated rewards, had factors depending on the energy level, the number of food items in the bag, food consumption history, and the distance from home.

4.3 Experimental results

We compared three kind of reinforcement learning techniques to test the benefits of factorisation:

1. Q-table based learning (SARSA(λ)), no factorisation, only state-action values are learned, which implicitly incorporate transition probabilities.

Algorithm 1 : Agent life cycle. The agent applies temporal difference learning with linear function approximation extended with (factorised) model learning and memory map based preprocessing to generate states from observations

inputs:
state variables $\{X_i\}_1^m$, actions A,
local function scopes (index sets) $\{\Gamma_i\}_1^m$, $\{I_i\}_1^k$ and $\{J_i\}_1^n$ for transition probabilities, reward function and value function

1: **for** each time step t **do**
2: collect primitive observations, update long term memory maps
3: observe reward r_t for previous action or state transition
4: generate current state x_t (high level variables) using memory maps
5: update value approximation according to Eq. 2.7 or using TD(λ):
$w_{t+1} = w_t + \alpha\ F(x_t)[r_t + \gamma V(x_t) - V(x_{t-1})]$
6: update transition model parameters based on the observed state transition:
increase frequency counts for variable values in $x_{t-1} \rightarrow x_t$ upon a_{t-1},
recalculate probability values from frequency counts
7: update reward function approximation:
$u_{t+1} = u_t + \alpha\ G(x_{t-1}, x_t)[r - R(x_{t-1}, x_t)]$
8: choose next action:
$a_t = \arg\max_a \sum_{x'} P(x_t, a, x')[R(x_t, x') + \gamma V(x')]$
9: $t \leftarrow t + 1$
10: **end for**

2. V-table based learning (TD(λ)) along with the estimation of transition probabilities (P) and reward function (R) utilising tables. This step of factorisation separates state values form state transitions, i.e. from the effect of actions.

3. Factored learning (factored TD(λ)), where the V, R and P functions are factored utilising local scope functions.

Surprisingly, the state space described above already proved to be too large for the table based methods to make progress in learning in a reasonable amount of time. To make comparison possible, we had to reduce the state space to a minimal size; we reduced the number of discretisation intervals for some variables, and dismissed the feature 'distance from home' and the macro 'go back home'.

To show the learning process of the various methods, we calculated certain *learning curves* or *performance curves*: at each time step when the agent made a decision, we examined its energy level, and derived a series containing 1s and 0s, 1 meaning the energy level was between the two thresholds, 0 meaning it was not. By moving window averaging this series, this learning curve should tend to 1, provided that the agent learns to keep its energy between the two thresholds in a stable manner. We conducted the following experiments:

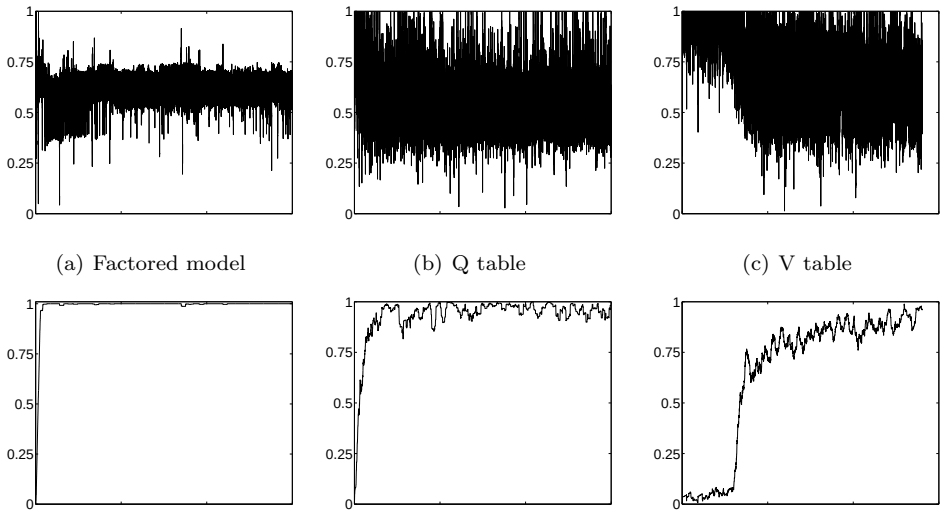
- We compared the energy curves of the agents during and after learning to examine how stable the behaviour of the agent was utilising the various methods.

High level variables	*Intervals or values*	*Notes*	*Action macro*
energy level	5	lowest and highest values are meant to be avoided	eat food, for each food type
number of food items in the bag	0-3	for each food type	collect food, for each food type
consumption history of food items	5	what fraction of the food consumed in the past few steps was of type t, for each food type	wait for a few time steps
distance to the nearest food item	5	for each type of food	explore; move in a random direction and amount
distance from home	5		go back home

Table 2: **High level variables and action macros used.** With these variables, (i) size of the state space is $5 \times 4^2 \times 5^2 \times 5^2 \times 5 = 250,000$, (ii) size of the state-action space is $5 \times 4^2 \times 5^2 \times 5^2 \times 5 \times 7 = 1,750,000$.

- We compared the learning curves of the various learning methods, to see how smooth the learning processes were, depending on the learning method.
- We tested how the various methods scale with the state space size.
- We tested the factored learning method in a slightly more complex setting, when we enabled the 'distance from home' feature and the 'go back home' macro. Also, in this setting the agent got penalised for being far from home, thus it had to optimise its behaviour according to two opposing objectives

In Figure 2 in the upper row, we show how the energy of the agent varies between the minimum and maximum values during a typical run for the three kinds of learning methods. The lower row shows the corresponding learning curves with a slight temporal smoothing. It can be seen that the factored model outperforms the table based methods both in learning speed and in its stability. The learning curve corresponding to the factored model goes up quickly to 1 right at the very beginning and stays there. The curve is smooth, demonstrating that the energy of the agent is kept between the two thresholds (which are 0.2 and 0.8). On the other hand, curves corresponding to table based methods rise towards 1 much more slowly and are much less stable, since the energy levels of the agents often exceed the thresholds. This comes from the fact that the factored model *generalises* very well, but table based models have no means to generalise, and need to learn the right actions for every possible combination of state variables.



(a) Factored model (b) Q table (c) V table

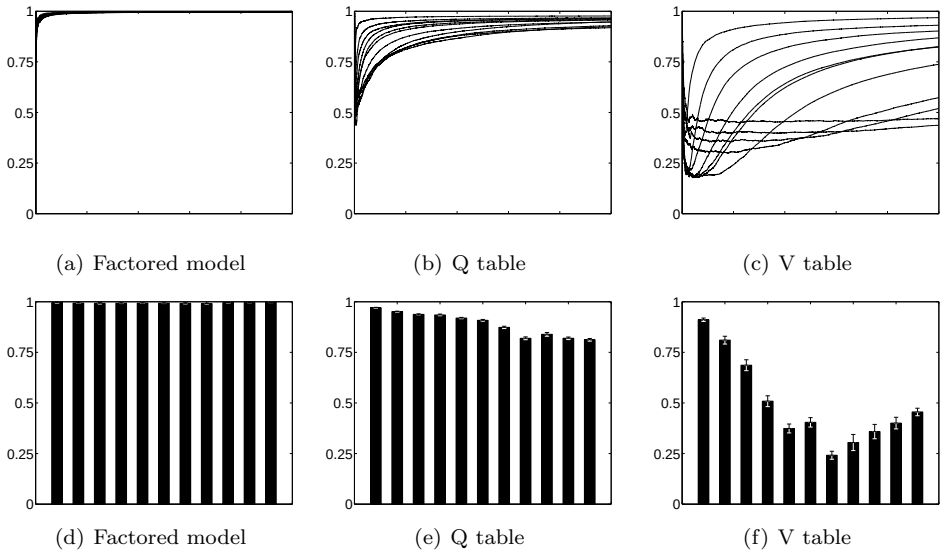
(d) Factored model (e) Q table (f) V table

Figure 2: **Typical energy levels and learning curves for the three learning methods.** The energy is the most stable for the factored model, for which learning is fastest and smoothest due to generalisation.

We also tested how the various methods scale with the state space size. To show the benefits of the factored approach, we increased the state space from minimal to a point until all methods could have been tested. In Figure 3, the curves are averaged over 10 runs and smoothed so that the learning trends could be seen. It can be seen that the learning time of the factored model is virtually not effected as the state space is increased, however, for table based methods, learning time increases greatly.

To see how the factored method behaves in a slightly more complex setting, we enabled the 'distance from home' feature and the 'go home' macro, and the agent also got punished based on its distance from home, to encourage it to stay near home, if possible. Note, that in this setting the agent had to optimise multiple criteria acting in opposite directions: to survive, it needs to get far from home, in order to get food, while at the same time it should spend as little time far from home as possible.

We examined the distribution of the agent's distance from home and concluded that it successfully learns to spend its time near home whereas it spent equal time at the two food areas when the feature was not enabled. In Figure 4(a) it can be seen, that if the agent is not punished for being far from home, it spends much time at the two ends of the world, which correspond to being close to home (one end of the world with one of the food sources) and being far from home (the other end



(a) Factored model (b) Q table (c) V table

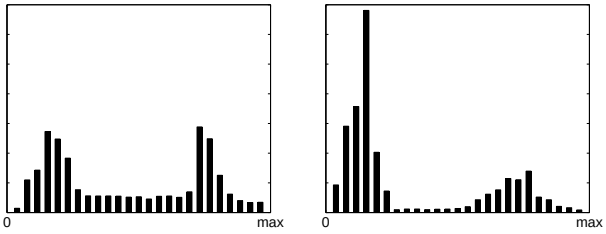
(d) Factored model (e) Q table (f) V table

Figure 3: **The effect of varying state space size.** Bar plots: averaged performances and corresponding standard deviations of the methods after 20,000 steps for state space sizes 1,620; 3,645; 6,480; 11,520; 20,480; 32,000; 50,000; 112,500; 381,024; 1,075,648; 2,654,208 corresponding to the bars from left to right. Upper figures: corresponding learning curves up to 100,000 steps. The factored model is barely effected by the increase in the state space size. Learning time of the table based methods increases steadily, especially for V tables.

of the world with the other food source), and it spends medium amount of time in the area between the two ends. On the other hand, if it gets punished for being far from home, it spends much time near home, and it spends much less time at the other end of the world (b). Although the agent must occasionally visit the other end of the world in order to obtain the other kind of food, it can also be seen that the time spent in the middle of the world also gets lower, which suggests that the agent learned to quickly rush to the other end, get some food, and return home.

5 Discussion

We have investigated a learning model based on factored Markov decision process in a task which is real world-like in two ways. First, our agent lives in grid world in which it observes only a small neighbourhood of its environment. This *partial observability* usually entails the fact that the decision process related to the observations is not Markovian, i.e. past observations are also required to make the appropriate decisions. Second, the space of observations is overtly large. So in



(a) Locations w/o 'home' (b) Locations with 'home'

Figure 4: **Locations of the agent, shown by distance from home.** The agent should go to the far end to optimise for the metabolism. It learns to spend less time at the far end and by 'travelling' if it is penalised for being further away.

a sense, there are too many observations which are still not informative enough. To enable decision making, the agent needs to form a Markovian description of its state. To do so, we have utilised hand coded high level variables for spatial and temporal integration of observations based on long term memory maps. These variables build up the state of the agent.

However, such a state space is still too large for tabular RL methods even for the simple task described in this paper. This points to the need for methods of other type that can take advantage of the structure of the state space. The factored model builds exactly on the characteristics that the state space is generated as the Cartesian product of state variables. We have compared the factored method to traditional table based RL methods. In real-world tasks the agent usually needs to learn the model of the task, i.e. the state transition probabilities and the reward function as well. Q-table methods, on the other hand, naturally incorporate model learning. In our studies, we investigated how the separation of those learning subtasks effects learning speed. Thus, we compared (i) Q-table based learning to (ii) V-table based learning augmented with model learning, and to (iii) factored value-learning augmented with factored model learning.

The experiments demonstrated that the separation of model learning from value learning in the tabular case corrupts performance, i.e. the V-table based methods augmented with model learning were always worse than the Q-table based methods that incorporate model learning into (state-action) value learning. This is probably due to the fact that the Q-table based method needs to learn a much smaller number of parameters, because it does not rely on transition probabilities. Transition probability tables scale quadratically with the size of the state space, and these parameters are elegantly cumulated into a much fewer number of Q values. However, when the model is factored, the number of parameters describing the transition probabilities and the state values becomes much less, and it becomes beneficial to separate the model from the values; learning speeds up because of generalisation. Our experiments show that the factored model learns very quickly and becomes

stable very soon, since it is able to generalise knowledge learned in one state to an other.

6 Conclusion

We have experimented with a factored version of temporal difference based reinforcement learning. The partially observable nature of the task was diminished by hand crafted features, which generated the factored state space. We have shown that factored learning is faster and more stable as compared to tabular methods. Furthermore, the factored method is also applicable to large state spaces since it does not suffer from combinatorial explosion. The capability that transition probabilities can be learned for the factored case opens the way for planning in complex situations, such as the environment of the New Ties project, including the development and the execution of joint plans about desired future states. This makes this method promising for realistic applications.

References

[1] Bertsekas, D. and Tsitsiklis, J. *Neuro-dynamic programming.* Massachusetts Institute of Technology, 1996.

[2] Boutilier, C., Dearden, R., and Goldszmidt, M. Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1104–1111, 1995.

[3] Boutilier, C., Dearden, R., and Goldszmidt, M. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1–2):49–107, 2000.

[4] de Farias., D. P. and van Roy, B. Approximate dynamic programming via linear programming. In *Advances in Neural Information Processing Systems 14*, pages 689–695, 2001.

[5] Dechter, R. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85, 1999.

[6] Dolgov, D. A. and Durfee, E. H. Symmetric primal-dual approximate linear programming for factored MDPs. In *Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics*, 2006.

[7] Guestrin, C., Koller, D., Gearhart, C., and Kanodia, N. Generalizing plans to new environments in relational MDPs. In *Eighteenth International Joint Conference on Artificial Intelligence*, 2003.

[8] Guestrin, C., Koller, D., Parr, R., and Venkataraman, S. Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468, 2002.

[9] Koller, D. and Parr, R. Policy iteration for factored mdps. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 326–334, 2000.

[10] Kveton, B., Hauskrecht, M., and Guestrin, C. Solving factored MDPs with hybrid state and action variables. *Journal of Artificial Intelligence Research*, 27:153–201, 2006.

[11] Liberatore, P. The size of MDP factored policies. In *Proceedings of the 18th National Conference on Artificial intelligence*, pages 267–272, 2002.

[12] Pineau, J., Gordon, G., and Thrun, S. Anytime point-based approximations for large POMDPs. *Journal of Artificial Intelligence Research*, 27:335–380, 2006.

[13] Sallans, B. *Reinforcement Learning for Factored Markov Decision Processes*. PhD thesis, University of Toronto, 2002.

[14] Sanner, S. and Boutilier, C. Approximate linear programming for first-order MDPs. In *Proceedings of the 21th Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 509–517, 2005.

[15] Schweitzer, P. J. and Seidmann, A. Generalized polynomial approximations in markovian decision processes. *Journal of Mathematical Analysis and Applications*, 110(6):568–582, 1985.

[16] Sutton, R. and Barto, A. G. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[17] Szita, I. and Lőrincz, A. Factored value iteration converges. *Acta Cybernetica*, 18(4):615–635, 2008.

[18] Szita, I., Takács, B., and Lőrincz, A. Epsilon-MDPs: Learning in varying environments. *Journal of Machine Learning Research*, 3:145–174, 2003.